NOTE: This paper has been submitted to ICSES; any reference should so indicate.

Technical Report #85-08

REQUIREMENTS FOR

SOFTWARE ENGINEERING LANGUAGES*

Arndt von Staa
Pontifcia Universidade Catolica
Rio de Janeiro, Brasil

Peter Freeman
University of California
Irvine, CA, USA

January, 1985

85-08

* Support for this research has been provided by IBM Brasil and IBM Corporate Technical Institutes, National Science foundation (US) grant MCS-83-04439, CNPq (Brasil) grant 40.16336/84, and FINEP (Brasil) grant 62.84.0416.00.

ABSTRACT

This paper analyzes the concepts of software construction embodied in the Draco system. The analysis relates specific mechanisms in Draco to particular software engineering (SE) principles and suggests future research needed to extend the approach. The purpose of the analysis is to help researchers understand Draco better and thus be able to direct in productive directions future research on this type of software engineering tool.

I. CONTEXT

Draco [Neighbors, 1980, 1984] is a tool to aid in the construction of similar software systems from reusable software components. It has been built and the principles of its usage are being developed in the context of a broader effort to improve our ability to reuse previous software engineering results [Freeman, 1983a]. This part of the paper establishes the context in which Draco exists.

THE DRACO TECHNOLOGY

The purpose of Draco, as illustrated in Figure 1, is to permit the capture and reuse of analysis information about real-world processes and design information about systems that automate those processes. The Draco technology consists of several notions from computer science and software engineering that are described in more detail in [Neighbors, 1980, 1984] and that are analyzed in Part III of this paper. This technology is embodied in a tool (current version is Draco 1.2) for building software systems from pre-existing software components.

Most of the concepts in Draco (we will not consistently differentiate between the technology and its instantiation in the tool) are well-known in computer science and include:

- meta-compiler techniques
- high-level languages
- refinement into lower-level languages
- modeling domains
- source-to-source program transformations

- software components.

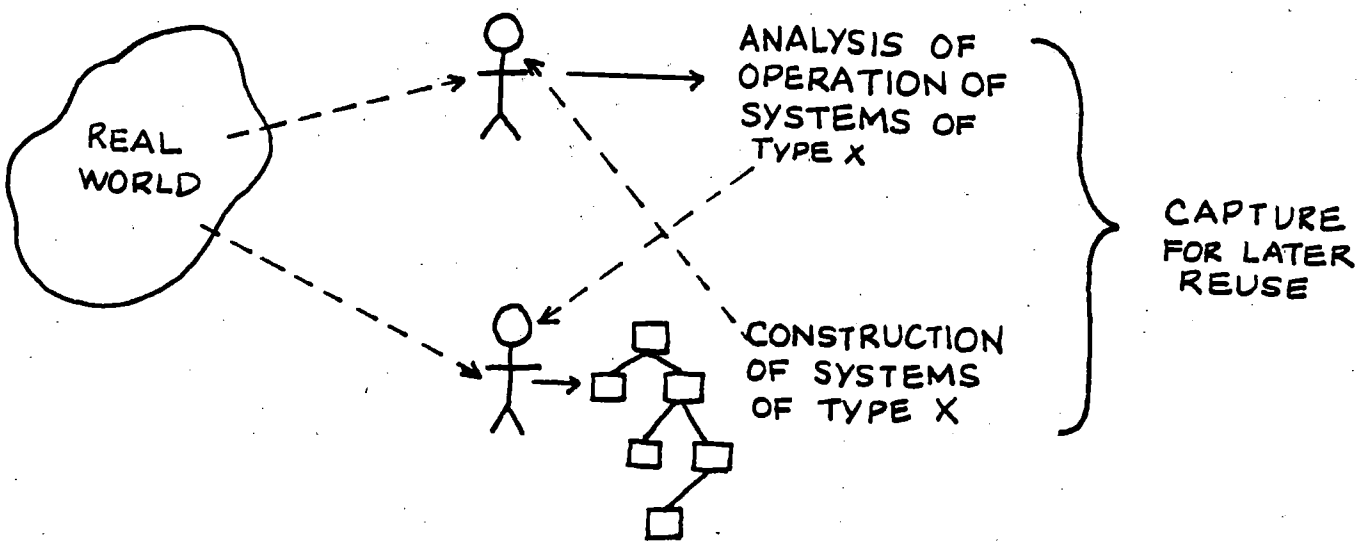


Figure 1: Purpose of Draco

While these concepts have been previously explored and utilized in a number of ways, the development of Draco in its present form brings them together in a new and productive way. The primary purpose of this paper is to analyze the Draco technology and its relationship to software engineering objectives.

A set of concepts relating to the reuse of any software engineering workproduct provides a larger context for the work on Draco. The next section describes this and Draco's relation to it.

REUSABLE SOFTWARE ENGINEERING

The twin forces of escalating costs and demands for improved quality have been driving the creation of new software development technology over the past

15 years [Buxton, Naus and Randell, 1968, Boehm 1973, DeRoze and Nyman, 1978]. While the primary driving force has been (and probably still is) the high cost of initial system development and subsequent evolution, the demand for increased product quality in software-intensive systems is emerging as an equally important force. Reusable software engineering [Freeman 1980, 1983a] is one attempt to address these problems.

Figure 2 presents the fundamental concept of reusable software engineering: Any software engineering activity (the analysis of requirements, creation of a design, planning of a series of tests, and so on) should be organized and based on principles that permit and encourage two specific sub-activities -- the explicit use of information developed in a previous instance of the activity and the

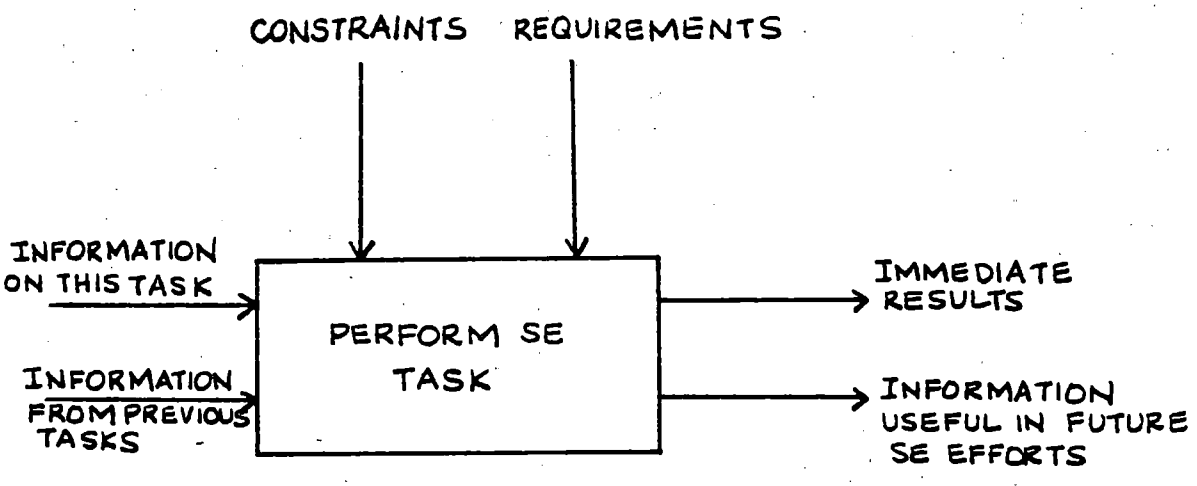


Figure 2: Central Idea of Reusable Software Engineering

creation of information (in addition to the primary output) that may be reused in a *later* instance of the activity.

It is easy to generate hypothetical examples of reusable software engineering: producing modular designs of a system that are later reused for implementation in a different language, developing a standard analysis of a problem area that can be applied to a number of applications, using a test plan that is obtained from a reference book for systems of a certain type. It is much harder to find actual examples of reuse. (However, it should be noted that the concept of code reuse, while quite old, is finally catching on [Lanergan and Boynton 1978, DoD 1983].)

The primary objective of reusable software engineering is to reduce the system-lifecycle cost and improve the quality of systems. We and others are working toward fulfilling this objective in specific ways. Draco can be viewed as a tool for the reuse of analysis (by capturing understanding in domain-specific languages) and the reuse of design (in the components that implement the languages). Thus, a broad understanding of Draco should include this larger context that motivates our work on it.

PAPER OVERVIEW

Our primary objective for this paper is to analyze the Draco technology and place it in a spectrum of software engineering/computer science concepts. This will permit further, rational development of the Draco concepts. While practitioners may be interested in it, other papers will address that audience more directly.

Neighbors' thesis has been widely distributed and still provides the most detailed published description of the Draco technology. Neighbors [1984] provides a summary while [Neighbors, *et al* ,1984], and [Arango and Leite, 1984] provide internal information on Draco. These detailed descriptions will not be repeated here.

Our paper has five major parts and an appendix. Part II provides an overview of what Draco does, explains the operational context in which it is intended to be used (which is further detailed in Appendix I), defines several key roles involved in its usage, and discusses its operational status and results to date. Part III analyzes Draco from the standpoint of the primary mechanisms on which it is based and their relationship to SE conceptual objectives. Parts IV and V discuss our interpretation of results and suggest paths for further development of the Draco approach to software construction.

II. DESCRIPTION

In this part we will review what Draco does and how it does it [*cf.* Neighbors 1980, 1984, for more detail]. This description, though brief, should permit you to understand our analysis of the underlying technology even if you have not read the earlier papers.

WHAT DOES DRACO DO?

From the standpoint of software technology, Draco can be viewed as a system that provides two main functions: 1) The definition and implementation of languages of various types (properly viewed as specification languages) and 2) assistance in and partial automation of the task of refining a specification of a

desired system (given in one of the languages known to Draco) into another language (also known to Draco). Draco also provides assistance in optimizing the programs produced, managing the libraries of languages and their implementations, and performing other housekeeping details. Figure 3 represents the main functionality of Draco.

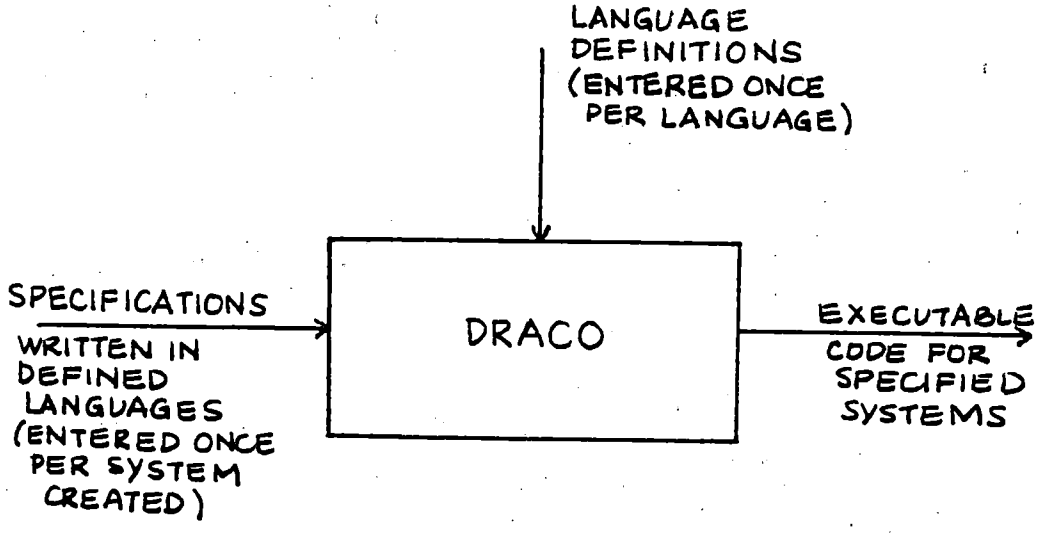


Figure 3: Primary Function of Draco

Before looking at how Draco works internally, your understanding will be aided by looking in more detail at *what* Draco does. Figure 4 expands on Figure 3, identifying more of the elements involved in using Draco to generate a system.

The use of Draco can be understood by studying Figure 4, starting at the top left of the diagram. If it is decided that it is worth the investment to provide Draco the capability of dealing with programs for a specific application domain, then a domain analysis is carried out by a highly skilled analyst (called

a *domain analyst*). A domain analysis is a generalization of the classical systems-analysis activity that precedes development of a computerized system: A systems analyst studies particular instances in which a computerized system is to operate, identifying the data elements and processes that are to be computerized. In the Draco realm, a *class* of situations is studied (Step 1) (each one of which might be a candidate for computerization) in order to identify all of the objects (data) and operations (processes) that exist in any situation belonging to this domain.

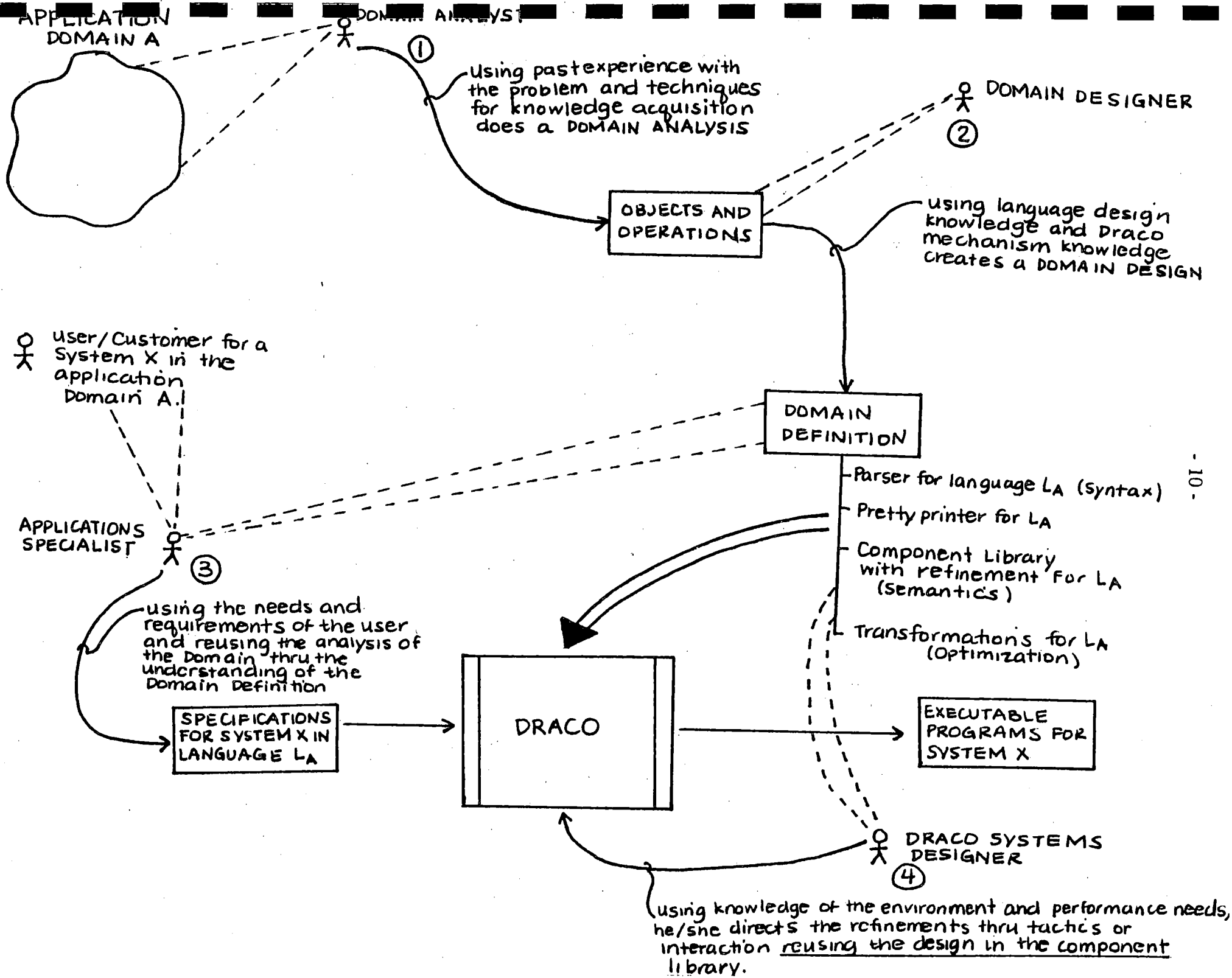


Figure 4: Organizational Context and Usage of Draco

The result of this domain analysis is a list of the objects and operations that are needed to specify systems in that domain.

The next actor that is involved is a *domain designer*, a language designer who also understands how to build the domain definitions that must be given to Draco. He or she then takes the domain analysis (the list of objects and operationcoupled with their semantic definitions) and defines a language incorporating them (Step 2) which forms the external aspect of a domain definition. It is sufficient to think of this as a high-level specification language for systems in that particular application domain. For example, we have domains for the specification of systems that process standardized school tests, implement decision tables, do numerical calculations, and so on (see below).

At this juncture (forgetting for the moment that the domain analysis and design activities may have to be reiterated just like any other design and analysis activities), Draco is ready to be used to help generate systems for use in this particular application domain. (It will also be able to help generate systems in other application domains that have been previously defined). We say that Draco is now operational in the A domain.

Suppose now that a *Draco applications specialist* (essentially a systems analyst who is expert in Draco) is called upon to create an A type system for some specific situation (not necessarily one of those originally investigated for the domain analysis). He or she will perform a traditional requirements analysis and definition conditioned by the existence of the specification languages available in Draco. Their next step, however, will begin to use Draco explicitly by specifying the system to be created using the domain language L_A for

application domain A (Step 3). This system specification is then fed into Draco by a *Draco systems designer* (who may be the same person) who is skilled in the technical use of Draco; this person guides Draco in the production of executable code (Step 4). The executable code that is produced by Draco under the guidance of the systems designer is then ready for installation in the user environment.

We have glided over a number of important and in some cases difficult aspects of systems development using Draco. For example, what if the domain language is not sufficient for expressing the desired application? What if Draco cannot produce a system because of constraints encountered in producing executable code, even though a syntactically legal input specification is given? What is done if changes are desired in the target system at some later date? These and a number of more technical questions are dealt with later in this paper or in other documents or, in some cases, are still open research questions. For the present, however, this idealized and error-free usage pattern will suffice for our discussions here.

Figure 5 summarizes the steps of Draco usage. After a domain is created (and debugged) Draco is used a number of times (often in parallel) over a long period of elapsed time to develop specific application systems of a particular type. This figure should actually be replicated a number of times, one for each domain.

Beyond its functional operation, it is important to understand the organizational context in which Draco is intended to be used; this is illustrated in a

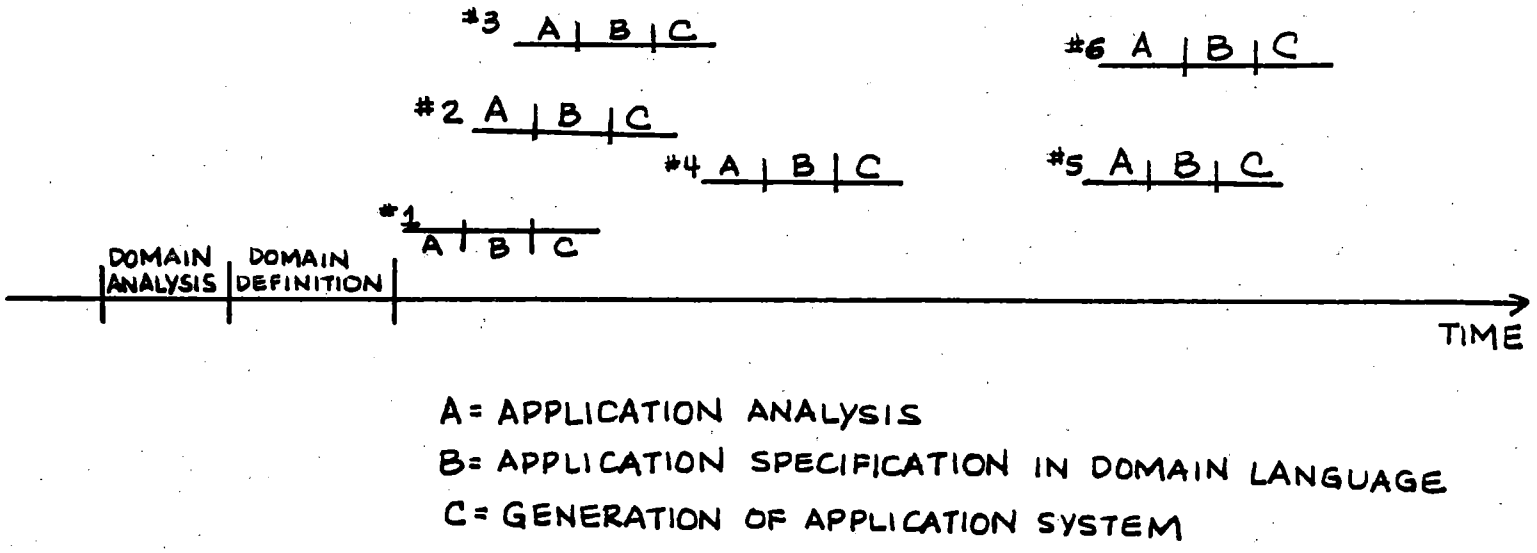


Figure 5: Summary of Draco Usage

partial SADT model in Appendix 1.* Several assumptions about Draco must be made explicit:

- Draco is intended for use in situations in which numerous, similar systems will be created over time (e.g. payroll systems, project scheduling systems, educational programs, etc.);
- For a given application domain (e.g. statistical report generation) an analysis of this domain must be made and defined to Draco before it can be used to generate programs in the domain;
- It is not a fully automatic program generator, but rather aids the system creator in an interactive manner. (The amount of interaction can be kept small, however.)

*Due to Neighbors [1980].

OPERATIONAL RESULTS TO DATE

Draco was first developed by Neighbors [1980], further improved by him in its implementation [Freeman, 1984] and is still a research vehicle for exploring a number of concepts. Ultimately, it must be possible to measure its success against regular system development procedures. The largest system generated so far consists of approximately 1000 lines of LISP code produced from about 100 lines of specification. At this stage, however, it is not a production tool and results must be measured more qualitatively.

Key to the eventual success of the Draco approach is the existence of an effective library of domains that can be built upon. Examples of domains built so far include:

1. **Fully developed domains** (parser, prettyprinter, transformations and refinement library exist)+:
   - ATN: Augmented Transition Networks
   - DIC: Dictionary construction
   - GEN: GENeration of sentences in natural language (subset parsed by the ATN domain with words in the DICtionaries generated by DIC.
   - TASK: Limited facilites for simulation parallel processing
   - RML: Reuse Module definition Language
2. **Partially developed** (lack transformation and refinement libraries):
   - I8080: CP/M, assembly Intel 8080 domain
   - ASM: A general assembly language - low level operating system language
   - PL: Simple programming language for demos and testing.
   - STA: Standardized Test Analysis
   - TDG: Tactical Display Generation
3. **Domains** that pertain to the Draco "infrastructure" (fully operational):
   - PARSER Generation

+All of the domains in class 1, with the exception of RML, were developed by Neighbors in his thesis.

TACTICS Specification
Refinement Libraries Construction
Transformation Libraries Construction
Prettyprinter Generation

Discussion of the domain construction aspect of Draco is quite involved because of the complex nature of the task and the connections to several other areas (artificial intelligence, language design theory, data modeling, and so on). This topic is the current focus of our research, but will not be further discussed here.

Likewise, the operational aspect of Draco (its physical construction) needs to be thoroughly analyzed and understood. This, too, will be treated elsewhere.

HOW DOES DRACO WORK?

Pursuing the model of Draco given above -- a black box that takes system specifications written in high-level languages and produces executable (or compilable) code -- we need to indicate briefly how this is done. Bear in mind that there are two distinct phases of Draco operation: *domain set-up* and *specification refinement.*

The domain set-up phase is when a domain language is defined to Draco in an operational way. During this phase Draco employs meta-compiler techniques similar to the META-II procedures described in [Schorre, 1963]. The primary thing of note in this phase of operation is the fact that the domain designer has available several languages (PARGEN, PPGEN, TFMGEN, REFGEN) to define new domains to Draco. We will not be further concerned here with the domain-setup phase.

In order to understand the operation of Draco in the specification refinement phase, let us look at its internal logical structure. Figure 6 portrays the four elements of a domain inside Draco: a parser and a pretty printer for the domain language, a set of source-to-source transformations on the language

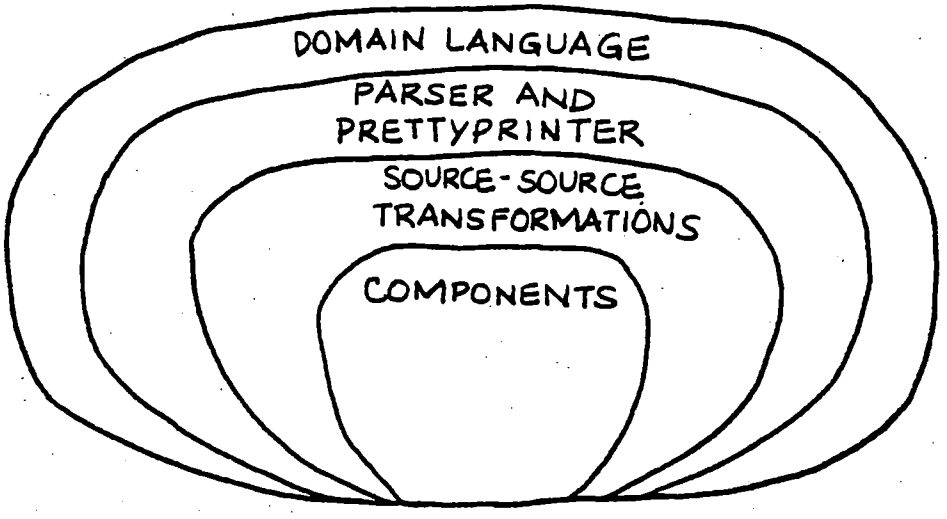


Figure 6: Elements of a Domain

used to optimize or otherwise change expressions in the language, and a set of components that provide the semantics of the language.

These components are the key to Draco's ability to generate code. Essentially there is a component for each type of operation and type of object in the domain language. Each of these small programs (or set of data declarations) may be written in *any language known to Draco*, including the language of the domain itself. There may, in fact, be several alternative components for a given

element of the domain language (we are using "domain language" and "specification language" interchangeably) which can be chosen depending on the objectives of a particular system construction task. We say that the component provides a *refinement* of the particular construct in the language into the other language.

The transformations are used to change the form of an expression in a domain language either to gain a desired performance characteristic (*e.g.* greater accuracy at the expense of speed) or to put it in a form more amenable to refinement into another domain. While quite important, it is not necessary to our purposes here to describe further this aspect of Draco's operation.

The basic logic of Draco's specification refinement mode of operation can now be easily described: Given a system specification in a particular domain language, appropriate parts of the specification are chosen for refinement and a refinement applied. Assuming this can be done (no constraints are violated) this refinement results in a new specification in which some parts of the original specification have been logically replaced by their refinements. This process is iterated until the entire specification is expressed in the desired target domain (usually one whose language is executable) or appropriate refinements cannot be found. This logic is displayed in Figure 7.

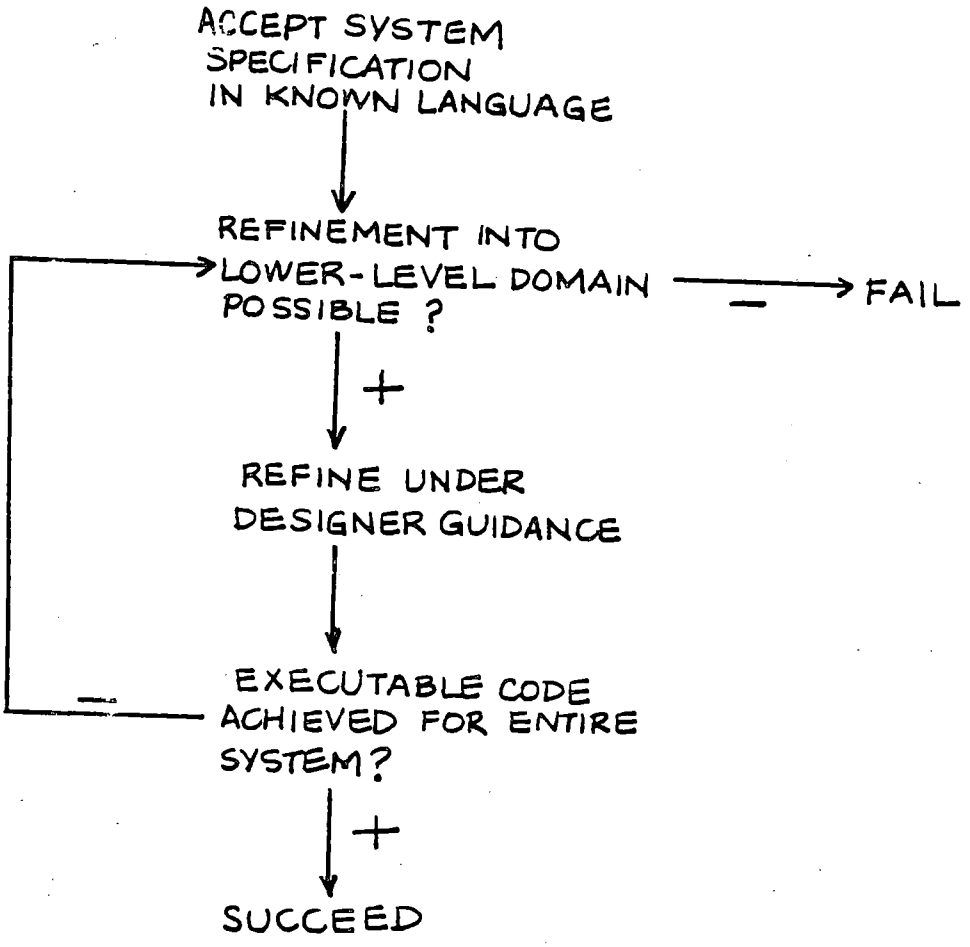


Figure 7: Logic of Draco's Operation

We must emphasize that this description of Draco's operation omits considerable detail. It is sufficient for a casual reading of the analysis that follows, but for a deeper appreciation of the analysis the references should be consulted (especially [Neighbors 1980, 1984]).

III. CONCEPTUAL ANALYSIS OF DRACO

The previous part has reviewed the operation of Draco; in this part we will analyze Draco conceptually and relate its observed capabilities to some important principles of software engineering.

Artifacts are often described as though they have been derived from principles in a top-down fashion. As most of us are well aware, that is seldom the way things happen. Thus, it is important to understand that what we are doing in this section is analyzing and trying to explain the reasons underlying Draco in an *ex post facto* manner, not describing precisely how it was created in the first place. (However, that does not mean that these principles were not explicitly used).

We begin this section with a brief discussion of software engineering objectives and then follow with our analysis. The analysis centers on showing the relationship between six important SE objectives and four primary mechanisms found in Draco.

CONCEPTUAL CONTEXT OF DRACO

Software engineering is concerned with a number of objectives, including:

- meeting current customer/user needs economically;
- increasing system quality;
- providing tools to increase the intellectual capabilities of system developers in order to meet future needs.

A number of approaches are being pursued to reach these objectives, including the creation of software engineering environments, the application of artificial intelligence techniques, retraining of personnel, development of automated tools,

utilization of mathematical techniques, and so on. The Draco technology cannot be completely contained in any one area (just as many of those areas in fact overlap) and it would be a mistake to do so. (It is important to remember that we identify "areas" primarily for pedagogical or professional reasons).

In general terms, Draco is a long-range attempt to build a production-quality tool for the working software engineer. It incorporates important concepts from several areas, including language design, language translation, information modeling, specification methods, heuristic techniques, and automatic programming. It is anchored in the traditional system specification and code production phases of the lifecycle so that it does not totally change the common SE pattern. Its use of AI (such as pattern-matching techniques and semantic networks) is important, but may not be recognizable by AI researchers; thus, we would not stress that area.

Its relationship to the area of resuable software engineering is that it provides a mechanism whereby we may reuse the results of analysis and design more easily. Specifically, we are able to capture and reuse a deep understanding of an application area (analysis information) through the design and utilization of domain languages.* Design knowledge is captured in the components that provide the semantics of the domain languages.

The following sections will make these connections more explicit.

---

*The discipline of analyzing a domain sufficiently to permit the design of a language may yield considerable benefit even if it is not actually used in Draco.

CONCEPTUAL OBJECTIVES

We listed several objectives of software engineering above. These are utilization, or *external*, objectives of SE in that they relate to the utilization of software engineering in the broader context of creation of software-intensive systems for some larger reason. In working with SE as a discipline, we usually focus on a set of *internal* concepts that we want SE techniques to possess. These desirable conceptual properties of a SE artifact can be thought of as objectives for Draco to reach.

Another way of looking at it is that there are an established set of properties that people believe the practice of software engineering should have in order to attain the utilization objectives mentioned above. For example, it is generally accepted that if a SE technique helps us use levels of abstraction that this will advance us toward our goal of dealing with complexity.

Our primary mode of analysis in this section is to establish a set of conceptual objectives that we want to attain or have extant and then show how Draco contributes to those objectives. Our purpose in doing this is to evaluate if, and in what ways, Draco is a useful SE tool. (Attainment of the utilization objectives of lower cost, and so on, will provide the definitive answer to this question; however, that analysis takes much longer to achieve and does not provide information on the source of the power of the tool).

The six conceptual objectives we have chosen to anchor our analysis are:

Abstraction levels: Being able to abstract from a situation and deal only with a name that stands for a particular piece of the complexity of the situation is a key concept in almost all of SE. Coupling that with hierarchy

is a standard and necessary aspect of systems work.

Structure of products: The software artifacts that we create must have structure to permit us to modify them, measure them, and fit them with other systems. A goal of most SE work is to produce well-structured systems. This concept is usually interpreted in a broad fashion so that it includes a number of definitions of "good" structure as well as the structure of workproducts other than code.

Systematic processes: A key principle in all aspects of software engineering is that of doing things in a systematic, well-ordered way.

Modeling: The invisibility of software forces us to do all of our SE work through the manipulation of models of the actual machine states we wish to create; this basic fact extends all the way back to the requirements analysis activity in which we build models of real-world situations.

Compartmentalization of knowledge: One of the primary contributors of complexity is the fact that at any point in development there are many separate pieces of knowledge or information that may potentially bear on the decisions that must be made. A successful strategy, borrowed from artificial intelligence, is to break the applicable knowledge up into subsets that permit one to deal with less information.

Reusability: Standard engineering practice in other fields is to reuse parts, assemblies, tests, and techniques wherever possible. The same philosophy can be applied to software engineering as described above.

These do not form a complete set of desirable properties for a SE tool or method, of course.* For example, a currently important objective is for any new technique to be well-integrated with existing techniques in order to permit easy usage of results from one technique by another. We have not included that property here because Draco does not particularly contribute to it. We have included those that are most heavily related to the mechanisms of Draco.

It is also important to understand that there is not uniform agreement as to

*See [Freeman and Von Staa, 1984] for a similar, but somewhat more basic, set of principles.

the desirability of attaining these properties. For example, some would argue that utilizing systematic processes in the creation of a software system may unduly restrict creativity; indeed, this is a danger if the systematic aspect of SE is carried to an extreme. The current interest in prototyping stems in part from this viewpoint.

We will not try to argue here for the appropriateness of these particular objectives, but rather rely on their widespread acceptance as proof of their importance to software engineering. (The interested reader is referred to [Fairley, 1985], [Freeman, 1983b], and [Pressman, 1982] for discussion of their relevance). We assume, then, that the reader agrees that it is desirable to carry out a software engineering activity that has these properties. Our task in the remainder of this section is to demonstrate in what specific ways Draco contributes to these objectives.

ANALYSIS OF DRACO MECHANISMS

Figure 8 presents a summary of our analysis of how the mechanisms of Draco contribute to certain SE objectives. For each of four main mechanisms in Draco, their contribution to each of six specific SE objectives is analyzed. The remainder of this section presents our analysis by mechanism.

	MECHANISMS	USED	IN DRACO	
CONCEPTUAL OBJECTIVES	multiple hi-level specialized spec languages	components/ assemblies	source-to-source transformations	refinements
abstraction levels	provide references (names) to levels	implement levels	N/A	permit specifications at one level to be converted to another level
structure of products	facilitate structure of specifications matching structure of problems	traceability of individual parts	N/A	force structuring at the specification level
systematic process	provide explicit starting point	provide focus for QA & design assurance	permit optimizations in controlled ways	permit breakup of process into small steps
modeling	direct representation of objectives and operations to be modeled	provide instantiation of external objects	provide for customization of modeling structures	provide for change to different modeling levels
compartmentalization of knowledge	focus language on problem at hand	design knowledge captured in very small pieces	capture optimization information	permit knowledge to be grouped by level
reusability	captures analysis information	captures design information	captures optimization information	N/A

Figure 8: How Draco Mechanisms Relate to Conceptual Objectives

Multiple Hi-level Languages

Abstraction levels. Programming languages provide for a single level of abstraction in describing computations via subroutine or function calls; this somewhat rudimentary level in the past was often further reduced in effectiveness because of the restrictions in length and format of the identifiers that could be used. Some more modern languages (Ada, Modula) have improved this situation slightly by providing mechanisms for creating and naming new operations and operands; the naming capabilities, however, are still within the confines of the syntax and semantics of the host language.

Draco permits a directed graph of domain connections. This fully supports levels of abstraction since one can build up an arbitrary hierarchy of independent languages corresponding to whatever hierarchy of abstraction levels is desired. This provides for references to levels of abstraction.

For example, conceptually it might be desirable from a SE perspective to establish levels of abstraction corresponding to arbitrary algebraic computations, calculations on data sets of real numbers, statistical calculations, and statistical calculations on sets of readings from a particular class of sensors. Draco would easily provide for this through the creation of individual domain languages corresponding to each level of conceptual abstraction. Figure 9 portrays this situation and illustrates how the languages created might be utilized in the realization of several domains other than the one initially desired.

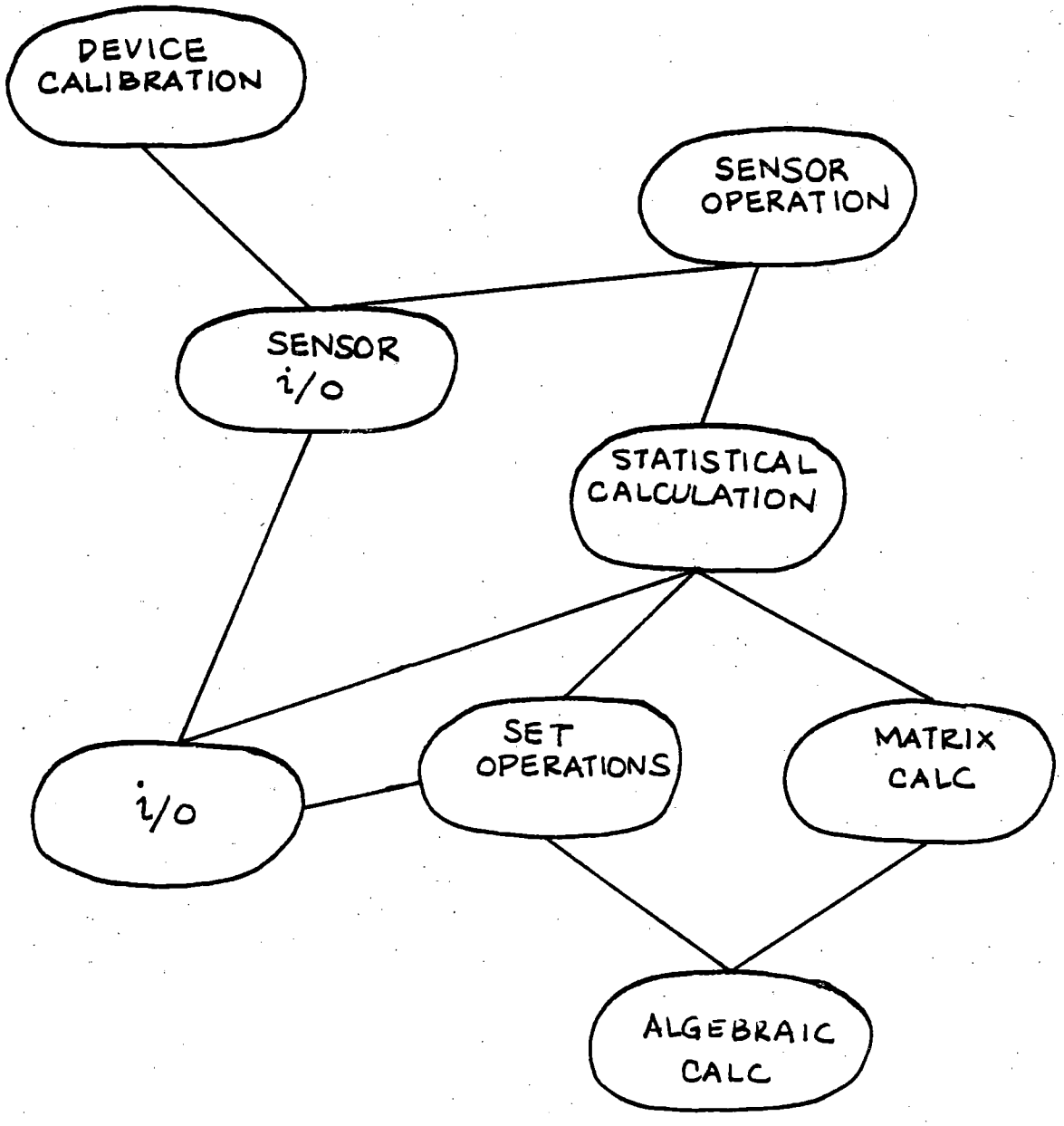


Figure 9: Example of Levels of Abstraction in Draco Domains

If one had to create an entire new language for each new level of abstraction, it would by a definite disadvantage.* This is the central reason that Draco

* As we gain more experience in creating new domain languages, it is becoming clear that one often wants to create a new language out of existing languages. For example, if you had a language for describing statistical calculations and another language for describing tabular report formating, you might well want to create a new language formed simply by "gluing" these two

is *not* well suited to one-of-a-kind development situations, but rather is intended to be used in situations in which the overhead of creating new languages (corresponding to new levels of abstraction) will pay off in multiple usage.

The advantage is the ability to create and provide the linguistic means for manipulating levels of abstraction that closely correspond to the "natural" levels of conceptualization that are dictated by the problem area and by the problem solving strategies chosen by the software engineer. Because of the importance of representation in design [Freeman, 1980] and problem solving [Rubinstein, 1975] and the centrality of levels of abstraction to almost all software engineering, this is very likely the most important contributor to the power of Draco.

Structure of products. The structure of the software produced using Draco will be impacted in two ways. The fact that Draco has languages closely fitted to particular application areas will provide strong support for the important structuring principle that the structure of the program written in the domain *language* should match the structure of the problem in the *domain*. For

---

languages together. In this case, the "glue domain" is formed relatively easily by taking the desired language elements from the underlying domains to form the new domain.

It should be noted that there are other approaches to this problem of creating a new language out of existing ones (which is different from using existing languages to implement the semantics of the new languages since in the glue domains we are carrying over the syntax as well). One approach is to establish specific interfaces in the syntax of the new language that tells the parser that the following language elements are to be parsed in a different context (domain). The disadvantage of this is that it leads to an N-square problem requiring interfaces between everything.

A second approach is to provide an explicit domain interconnection language for tying together existing domains. This approach may have some advantages, but has not yet been explored.

Because Draco does not have mechanisms for dealing with the interactions between domains at the level of language boundaries, we have pushed the problem of gluing domains together outside of Draco itself and turned it into a language design problem. The designer of a glue domain has the responsibility to create a single notation to handle the syntax of both underlying domains. While limited thus far, our experience indicates this is an effective approach.

example, if the problem consists of application of three basic operators and seven suboperations to a variety of objects depending on certain conditions, this problem structure could be easily modeled in the Draco specification by a domain language that had those operations, operands, and control structures and no others.

The second impact is that the code produced by Draco will not have the same structure we have come to look for in "well- structured" programs. At a first glance, code produced by Draco may appear to be highly unstructured. However, there is a very definite structure in it that is dictated by the components used to produce it rather than by the control structures of the final implementation. While this will make the code difficult to work with by hand, the intended modification procedure is to alter the specifications (or components) that led to the code and let Draco regenerate it.

Systematic process. The overall philosophy of Draco provides and enforces a well-defined and systematic process for creating a system. The contribution of multiple languages specifically is to provide an explicit representation for the starting point of development, no matter the area (assuming Draco can deal with that particular area). This is a very important contribution because one of the softest areas of development is often the starting point because of the absence of any way of representing the specifications. The necessity for representing all of the information for a particular development thus enforces a systematic process right from the beginning.

Modeling. The specification languages of Draco provide for direct modeling of the objects and operations from the application domain of interest. While in

theory we can always model those entities with any computable language, the existence of specialized languages results in effective ways of carrying out the desired modeling. (Our current research, as outlined below, is centrally concerned with determing the degree of this effectiveness).

A potentially negative aspect due to this feature of Draco, however, is the fragmentation of representation. Because Draco utilizes very narrow languages (in application scope), typically one will have to utilize several languages for a specific modeling attempt. How to tie together several languages in an effective modeling effort is one of the primary questions currently being addressed; as noted above, our current solution is to create "glue domains."

Compartmentalization of knowledge. The narrow "width" of the languages of Draco automatically focus one's attention on a narrow domain of knowledge. This will help achieve the goal of compartmentalization *if* the boundaries of the languages correspond to "natural" boundaries of the knowledge. If they do, then the multiple-language feature of Draco will provide a very powerful support of the objective of comparmentalizing knowledge.

Reusability. One of the goals of reusability is to make it possible to reuse the understanding of a problem or problem domain developed during the analysis portion of development. The languages that can be created using Draco permit the capture and reuse of information about an entire problem domain. Specifically, the languages capture the domain analyst's understanding of which operators and objects are appropriate for describing operations in a particular domain. Later, when a system analyst picks up that domain language to use it, he or she is then reusing the analysis of the domain originally developed by the

domain analyst.

Components and Assemblies

Abstraction levels. Components are used to implement levels. The functionality of the components in a given level implement the functions of that level (and similarly for the objects of the level). Another way of looking at it is that the components provide the connections to other levels above and below. Since a component may be written in terms of objects and operations of a lower level, this provides the connection to lower levels; similarly, the use of the functions (implemented by a component) at a higher level provides an upward connection.

Structure of products. Components may not show up in the code produced by Draco directly because of the refinements and transformations that may be applied. However, through the refinement history [Neighbors, 1980] one can trace the influence of a given component on a specific piece of code (whatever it may be). This provides, then, for one of the primary benefits of well-structured code -- the ability to find a given functional piece for purposes of change or repair. Draco changes somewhat the traditional idea of well-structured products in this sense since now modifications would be made at the specification level and automatically traced down to the code level.

We could also view this as two forms of structure: Horizontal within a level of representation (e.g. code) and vertical structure between levels (e.g. between the higher-level specifications and the code).

Systematic process. The reliance on components provides a locus for the application of design quality assurance. Further, the size of the components

facilitates the assurance of their quality. From a SE standpoint, this should help make the process of building a system more systematic.

Perhaps more importantly, the overall philosophy of Draco enforces a strong systematization on the process since it forces most internal design decisions to be made once at domain design time, not repeatedly each time a system is built. This form of structuring of the process is not possible if monolithic (i.e. non-componentized) systems are built. In effect, by utilizing many small components, we are able to break up the design process in a way that permits us to carry it out once in many small pieces (the design of the components) and then reassemble that design process in new ways (semi-automatically) when the components are utilized.

Modeling. Components provide for the realization of the external objects and operations to be modeled in a straightforward one-to-one manner. Further, taking a vertical view, they provide for connection between different modeling languages.

Compartmentalization of knowledge. Components permit us to capture design knowledge (about how to implement specific operations and objects) in small pieces, as noted above. They permit us to encapsulate this form of system knowledge in minimally small chunks thus focusing our attention. Assemblies of components (that result from their use in a higher-level domain's components) then permit us to expand the size of the chunk as desired. Libraries of components capture further design information by storing alternative refinements for operations and objects.

Reusability. Components are the mechanism that permit us to capture and reuse design information. Each component encapsulates a designer's knowledge about how to implement a particular structure or function. Indeed, because components provide for alternative refinements (implementations), there is the possibility of capturing and easily reusing design tradeoff information that is not normally available in a design representation.

It should be noted, however, that there is a level of design information that is not explicitly captured in this manner. That is the information about the overall structure of a particular system. For example, we may have a domain language (probably of a glue domain) that permits us to describe inventory control systems. That language, while providing a sufficient set of operators and oprerands to describe such systems, does not tell the system designer what the overall structure of an inventory control system should be.

Source to Source Transformations

Abstraction levels and structure of products. As used in Draco, transformations do not strongly contribute positively or negatively to either of these objectives. (However, transformations trade structure for efficiency in some cases, thus impacting traditional program structure).

Systematic process. Transformations are used to effect optimizations and changes in the representation of a specification to enable other operations. This will control optimizations and thus contribute, albeit in a minor way, to enforcing a systematic development process. It is the overall structure of the process that Draco supports that is the prime contributor to being systematic; in this

sense, low-level transformations are important contributors since they permit us to encapsulate and thus control changes to each stage of refinement (but not in a version-control sense.

Modeling. Transformations provide us the ability to customize a piece of the system representation so that we can then refine the representation into a different modeling domain.

Compartmentalization of knowledge. Transformations provide the physical encapsulation of optimization knowledge.

Reusability. Most of the information captured by transformations relates to optimization of expressions in the language, thus facilitating the reuse of such optimization information. While important, there is a higher-level of optimization information relating to the problem domain that we believe could (and should) be captured by transformations.

For example, it might be known that in a particular situation of inventory control, the calculation of expected inventory levels in the future could be greatly simplified. If that knowledge could be captured in the form of a transformation then if a specification were written and those conditions were present, the transformation could be invoked to obtain the simpler form of the statements; this would be a very valuable form of reuse of optimization information, but one that we have not yet achieved in Draco.

Refinements

Abstraction levels. A key element of the use of the concept of levels of abstraction in design is the way in which connections are made between levels.

Of particular interest is the way in which a structure at one level is mapped into an equivalent structure at the lower level, since this will have a great impact on the correctness of the final product. In this context, the use of refinements is central to the power of the Draco approach.

Refinements, of course, provide for the translation of a set of specifications expressed at one level of abstraction into equivalent specifications at lower levels of abstraction. There are several key aspects to the way in which this Draco mechanism contributes to our ability to utilize levels of abstraction. First, their form permits us to incorporate multiple refinements, giving us alternative connections between levels of abstraction in a controlled manner.

For example, normally we might want a particular computation in an application language to be refined to a functional language. However, we might also want a refinement available directly to a procedural language for use in special cases. Draco's usage of refinements permits this, providing a representation for the alternatives. This should be compared to the usual programming language situation in which direct references through a level of abstraction can cause difficulties or to the usual design situation in which normally there is no representation available for the alternatives.

A second very important aspect is that the refinements are done by Draco (with human guidance), thus providing more rigor and correctness in the actual refinement. This is one of the weak links in the usual top-down refinement of a design in which refinement from one level of abstraction to another introduces errors. However, Draco has no mechanism to force this and does not check preconditions for the refinements.

Structure of products. Most structuring of SE workproducts takes place at the level of the code (in the sense of structured programs); this is what we called "horizontal structure" above. While code produced by Draco has some definable structure, it is not the structure that obeys one-in, one-out rules. Draco thus forces "vertical structuring," or structuring of the specifications from which the code is produced.

Systematic process. Refinements are the main mechanism that permit us to break the development process up into small, well-defined steps in Draco. The process that is broken up, however, is not the traditional lifecycle (that is addressed by the overall Draco approach), but is the process of taking high level specifications and converting them into executable code.

Modeling. The contribution of refinements to modeling is to permit change of a model to a different level. In essence, this is the same as the conversion between levels of abstraction.

Compartmentalization of knowledge. This is another key area in which the mechanisms of Draco contribute directly to the desired SE objectives. Refinements, because they permit us to have languages at multiple levels, permit us to group knowledge by level of abstraction (distance from the machine) as well as by type of content (division into domains). For example, this permits us to suppress details of a particular computation or operation.

Reusability. Refinements permit the reuse of language elements at a higher level, but this does not directly relate to the objective of reusable software engineering as explained above.

IV. CONCLUSIONS

As noted in the opening of this paper, our primary purpose here has been to lay bare some of the connections between the mechanisms of Draco and principles of SE. Thus, there are no startling conclusions to be drawn, as might be the case if we had been analyzing an experiment. There are several points, however, that we believe are illustrated and supported by the analysis just given.

First, Draco provides an environment in which several important mechanisms can work together productively to achieve a collection of objectives. It is not just an engine for applying source-to-source transformations or a meta-compiler or a library for small pieces of code. Rather it incorporates all these plus more. Our analysis has looked at the individual contributions primarily, but scanning across a row in Figure 8, you can see how several mechanisms work together to achieve a given objective. For example, the various aspects of modeling are provided by the combined efforts of hi-level languages, components, transformations, and refinements.

A related point is that we can't really judge the validity of these mechanisms until we have truly operational results to report. While Draco currently provides some interesting prototype behavior [Neighbors, 1984] the analysis only shows connections, not utilization power.

While it may not be entirely evident, this analysis has left us with the distinct impression that current mechanisms such as those presented here are relatively powerful. For example, the transformation technology being utilized does not appear to be a hinderance to achieving the objectives listed. Another way of

expressing this is to note that we believe quite a lot of power can be obtained from current mechanisms and that perhaps we should put more effort on utilizing what we have (although not to the detriment of continuing the search for new structures in computer science).

In reflecting on the analysis as presented, it seems that the first three objectives presented -- abstraction, structure of products, and structure of process -- are relatively straightforward; that is, we know how to achieve them. This is supported by the fact that the characterizations in Figure 8 for those three objectives (for the contribution of languages, components and so on) are sraightforward and definite; we can readily understand them.

The next three, however, -- modeling, comparmentalization, and reusability are not so clear in their connections. Partly this is a result of the fact that they are more complex objectives; but, it is also the case that considerable work remains to be done on achieving them (in the context of Draco as well as more generally).

That leads us to the final conclusion: Dealing with knowledge (as represented by these last three objectives) is an objective that is far from being obtained in SE. Put another way, we believe this analysis highlights a more general situation; namely, that even the careful application of some powerful mechanisms intended to deal with knowledge manipulation raises more questions than it answers.

V. RESEARCH DIRECTIONS

Consistent with the purpose of the paper, let us briefly indicate in closing some needed research that will further the Draco technology:

- Specialized languages are only as effective as their capture of real-world knowledge. The connections shown in Figure 8 will be worthless if we cannot effectively create the languages in the first place. Though not dicussed in this paper, this is in our estimation the primary research need with respect to development of the Draco technology and the one on which we are concentrating. How to do domain analysis, how to partition knowledge into domains, how best to combine languages -- these are all questions needing a great deal of study.
- Neighbors [1980] pointed out in his thesis, and we strongly agree, that a potentially powerful application of transformations is to capture application-level changes in system (or problem) description to effect major economies in system generation. This issue has not been attacked in any systematic way.
- Refinement histories, enabling a replay of a refinement sequence, would provide an essential part of a production-quality Draco. How to use such histories effectively has not been explored at all. Likewise, while refinement tactics (that automatically guide Draco in making local refinement decisions) are operational, there clearly is a need for strategies that could be utilized to guide automatically entire refinement sequences; this, too, has not been explored.

There are a multitude of other open research questions relating to Draco technology, its operation, and its utilization. Indeed, the analysis presented here should serve to highlight and delimit the connections between specific mechanisms and objectives so that these questions can be readily determined by others.

ACKNOWLEDGEMENTS

A large debt of gratitude is owed to Jim Neighbors for creating such a rich collection of research questions! The members of the Reuse Research Group -- Guillermo Arango, Ira Baxter, Sylvia Karmanoff, Julio Leite, Chris Pidgeon, and Ruben Prieto-Diaz -- provided incisive comments on earlier drafts of this paper that materially improved it; special thanks to Julio Leite for Figure 4. The word processing acumen of Nancy Pomicter and Lynn Caverly greatly improved the visual presentation of the paper. Finally, the continuing support of the National Science Foundation of this research is very gratefully acknowledged.

REFERENCES

Arango, G. and Leite, J. *Draco Maintenance Documentation*, ICS Dept. University of California, Irvine, March 1984.

Boehm, B. W. "Software and Its Impact: A Quantitative Assessment", *Datamation* pp. 48-59, May 1973. Reprinted in [Freeman and Wasserman, 1983].

Buxton, J. M., Naus, P. and Randell, B. *Software Engineering Concepts and Techniques*, Petrocelli/Charter, 1976. From the 1968-1969 Nato Conference on Software Engineering.

Department of Defense. *Software Techniques for Adaptable Reliable Systems (STARS) Program Strategy*, April 1, 1983.

DeRoze, Barry C. and Thomas H. Nyman. "The Software Life Cycle - A

Management and Technological Challenge in the Department of Defense", *Trans. Software Engineering*; (4,4), July, 1978, pp. 303 -318.

Fairley, Richard E. *Software Engineering*, McGraw-Hill, 1985.

Freeman, Peter. "The Central Role of Design: Implications for Research," *Software Engineering: Research Directions*, Academic Press, pp. 121-132, 1980b.

Freeman, Peter. "Software Construction Using Components", *Final Report on MCS*-81-03718, ICS Dept. University of California, Irvine, April 1984.

Freeman, Peter. *Reusable Software Engineering: A Statement of Long-Range Research Objectives*, University of California, Irvine, Technical Report 159, November 1980.

Freeman, Peter. "Reusable Software Engineering: Concepts and Research Directions", *Proceedings of the Workshop on Reusability in Programming*, ITT, Newport, RI, September 1983, pp.2-16. Reprinted in [Freeman and Wasserman, 1983a].

Freeman, Peter. "Fundamentals in Design", in [Freeman and Wasserman, 1983b].

Freeman, Peter and Von Staa, Arndt. "Towards a Theory of Software Engineering", submitted for publication, October 1984.

Freeman, Peter and Wasserman, A.I. *Tutorial of Software Design Techniques*, 4th ed., IEEE Computer Society Press, 1983.

Lanergan, R. and Poynton, B. "Software Engineering with Standard Assemblies", *Proceedings of the ACM National Conference*, 1978.

Neighbors, James. *Software Construction Using Components*, ICS Dept. University of California, Irvine, Technical Report 160, 1980.

Neighbors, James, Arango, Guillermo and Leite, Julio. *et al. Draco* 1.3 *User Manual*, University of California, Irvine, April 1984.

Neighbors, James. "The Draco Approach to Constructing Software from Reusable Components", *Transactions on Software Engineering*, September, 1984.

Pressman, Roger. *Software Engineering: A Practioner's Approach*, McGraw-Hill, 1982.

Ross, D.T. "Structured Analysis (SA): A Language for Communicating Ideas",

IEEE Transactions on Software Engineering, Volume SE-3, Number 1, January 1977, pp. 16-34. Reprinted in *Tutorial on Software Design Techniques*, 4th edition, IEEE Computer Society Press, 1983.

Ross D.T. and Schoman K.E. "Structure Analysis for Requirements Definition", *IEEE Transactions on Software Engineering*, Volume SE-3, Number 1, January 1977, pp 69-84. reprinted in *Tutorial on Software Design Techniques*, 4th edition, IEEE Computer Society Press, 1983.

Rubinstein, Moshe. *Patterns of Problem Solving*, Prentice Hall, 1975

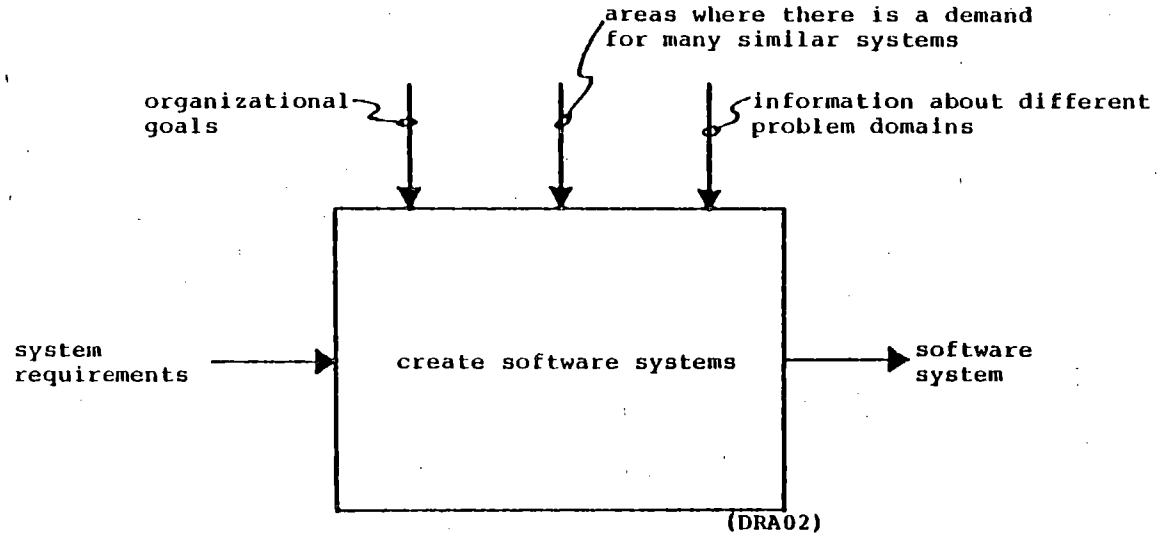
Schorre, D.V. "Meta II: A Syntax-Oriented Compiler Writing Language", *Proceedings of the ACM National Conference*, pp. D1.3-1-D1.3-11. ACM 1964.

APPENDIX I

MODEL OF THE USAGE CONTEXT OF DRACO

The SADT model presented in this appendix, taken from [Neighbors, 1980] describes in detail the organizational context of the intended usage of Draco. The model also illustrates the steps of using Draco that are described above in Part II. If you are not familiar with SADT, a description can be found in [Ross, 1977 and Ross and Schoman, 1977].

USED AT:	AUTHOR: James M. Neighbors	DATE: 15-OCT-80	WORKING	READER	DATE	CONTEXT:
	PROJECT: Draco 1.0	REV: 3	DRAFT			
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			
			PUBLICATION			

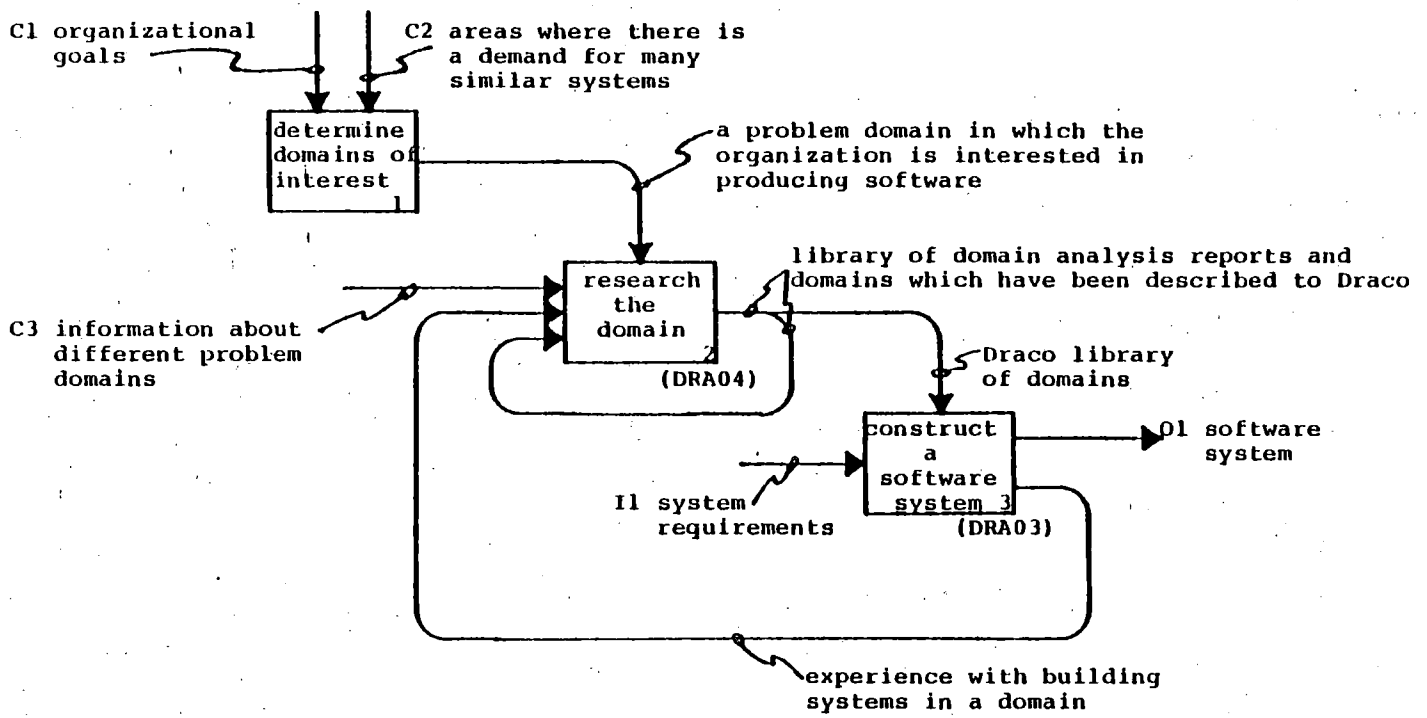


Purpose: To show the use of Draco within an organization which creates software systems.

Viewpoint: A researcher attempting to increase the productivity of the organization through the reuse of analysis and design.

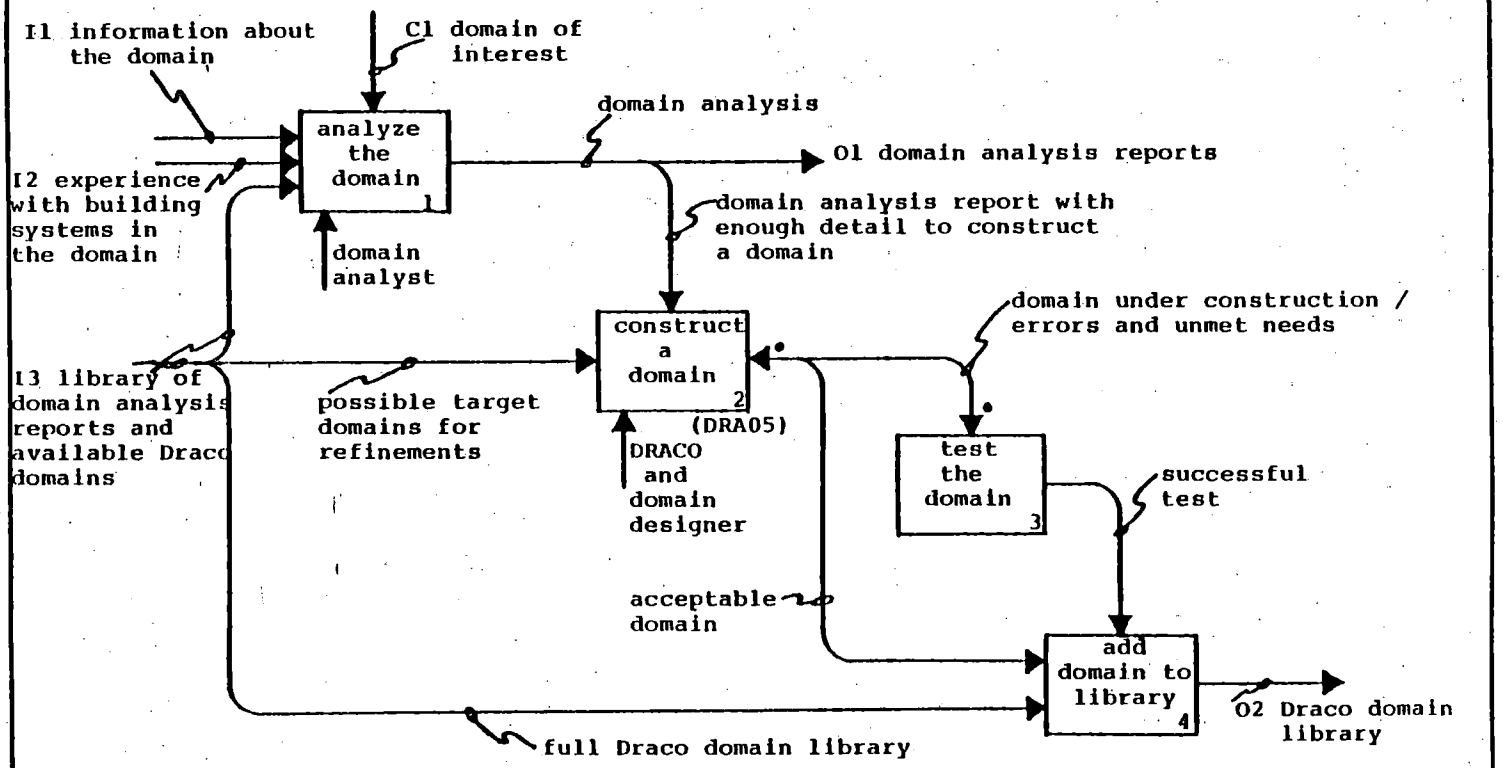
NODE: A-0	TITLE: CREATE SOFTWARE SYSTEMS (CONTEXT)	NUMBER: DRA01

USED AT:	AUTHOR: James M. Neighbors PROJECT: Draco 1.0 NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 15-OCT-80 REV: 3	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT:



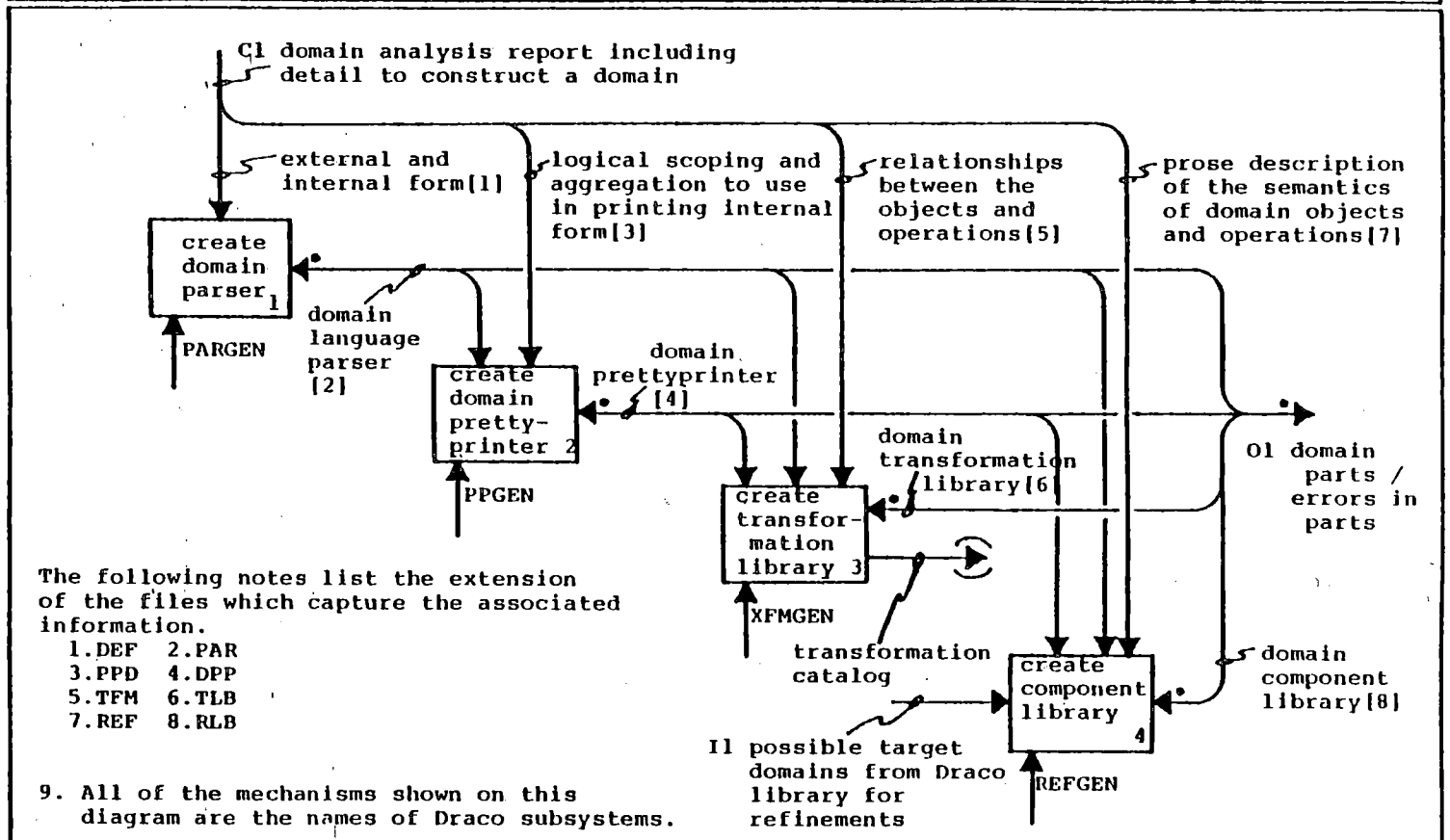
NODE: A0	TITLE: CREATE SOFTWARE SYSTEMS	NUMBER: DRA02

USED AT:	AUTHOR: James M. Neighbors PROJECT: Draco 1.0 NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: 15-OCT-80 REV: 3	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT:



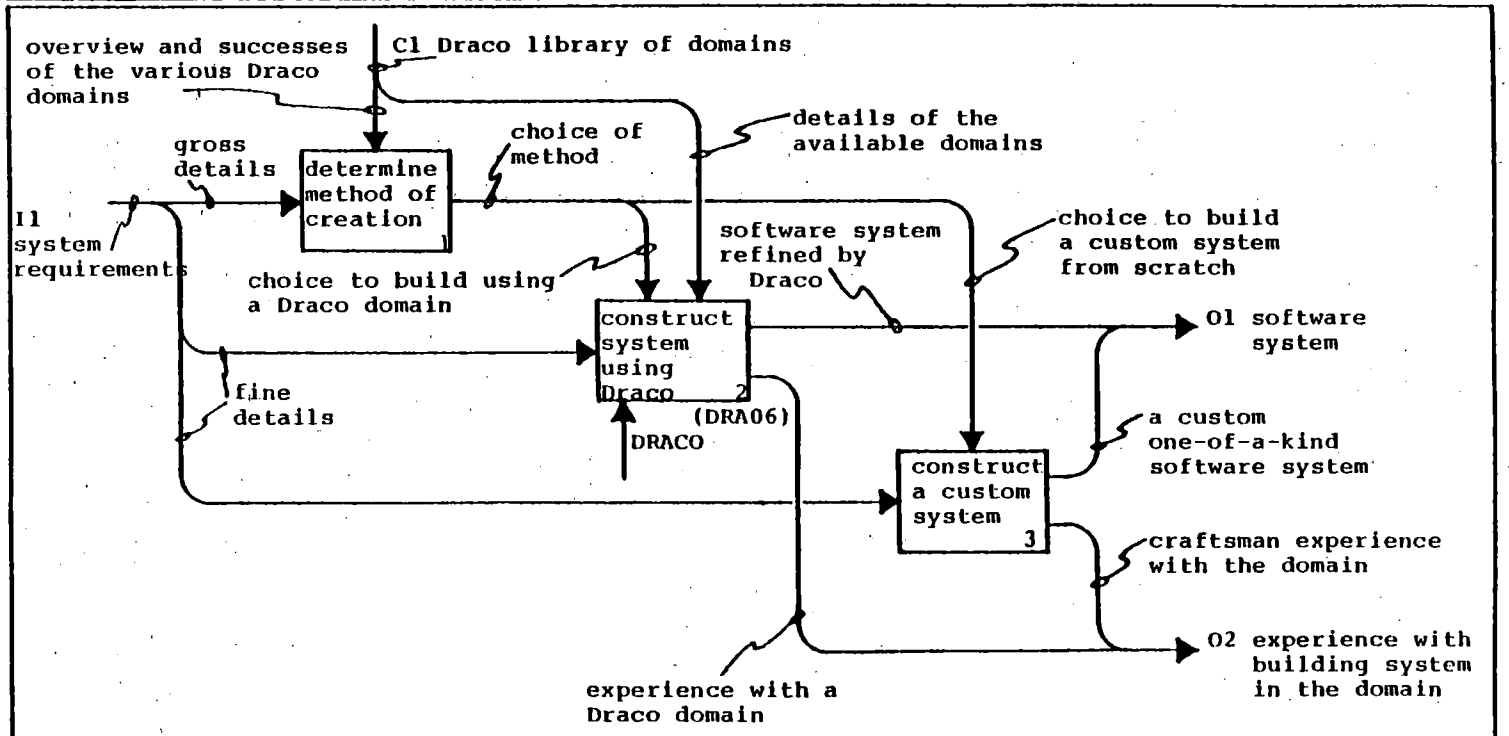
NODE: A2	TITLE: RESEARCH THE DOMAIN	NUMBER: DRA04

USED AT:	AUTHOR: James M. Neighbors	DATE: 15-OCT-80	WORKING	READER	DATE	CONTEXT:
	PROJECT: Draco 1.0	REV: 3	DRAFT			
	NOTES: 1x 2x 3x 4x 5x 6x 7x 8x 9x 10		RECOMMENDED			
			PUBLICATION			



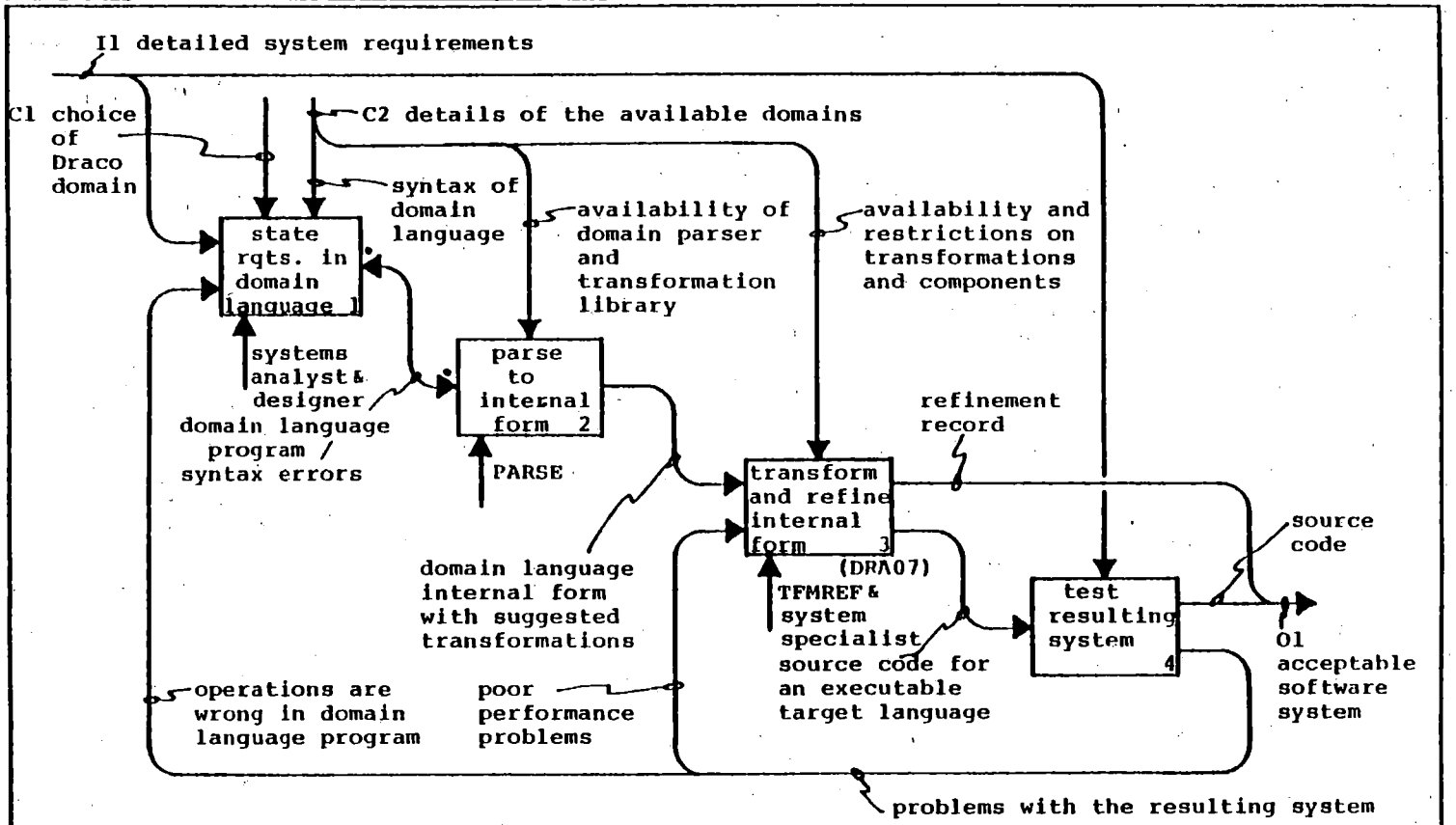
NODE: A22	TITLE: CONSTRUCT A DOMAIN	NUMBER: DRA05

USED AT:	AUTHOR: James M. Neighbors PROJECT: Draco 1.0	DATE: 15-OCT-80 REV: 3	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT:
	NOTES: 1 2 3 4 5 6 7 8 9 10				



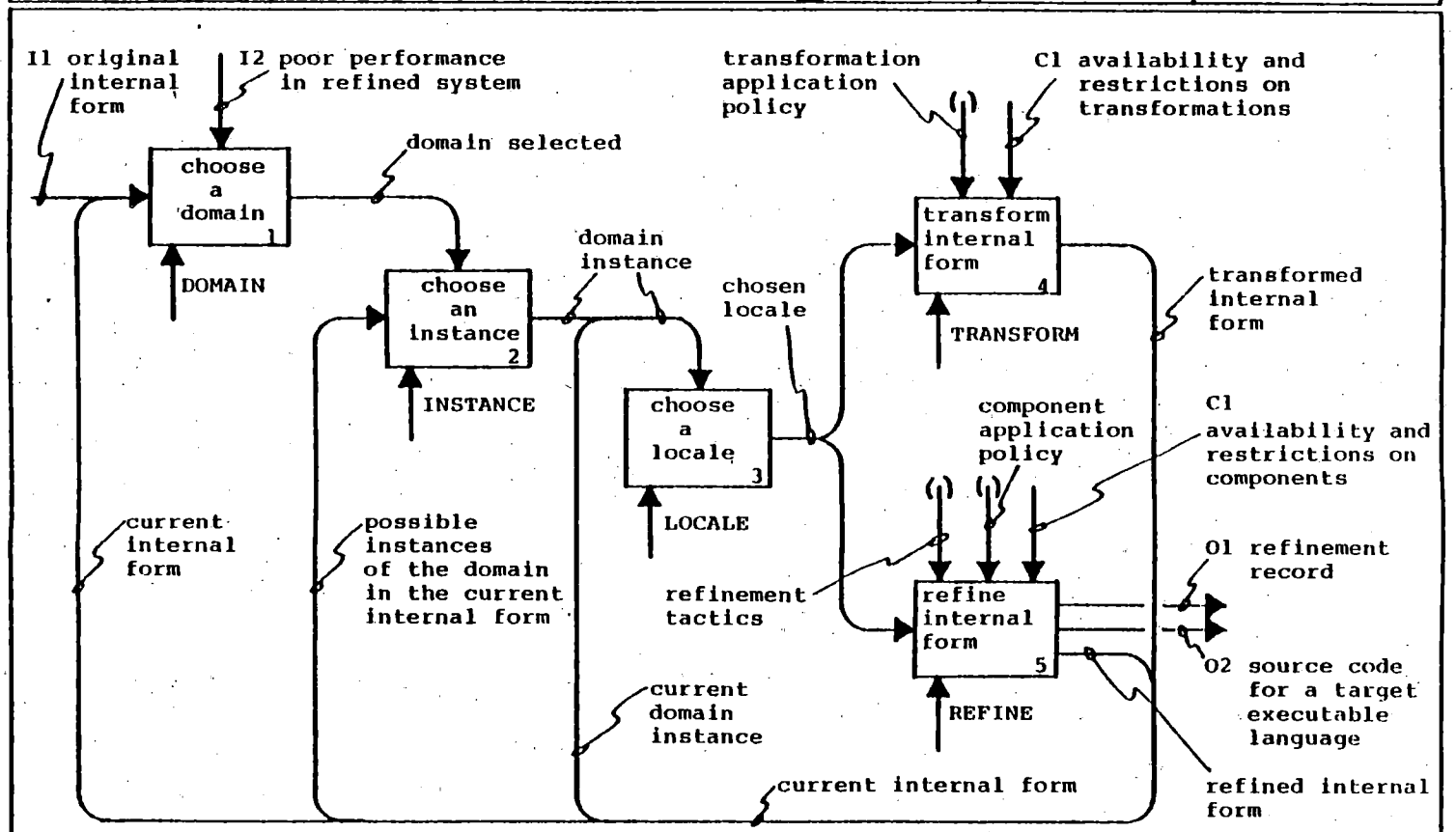
NODE: A3	TITLE: CONSTRUCT A SOFTWARE SYSTEM	NUMBER: DRA03

USED AT:	AUTHOR: James M. Neighbors PROJECT: Draco 1.0	DATE: 15-OCT-80 REV: 3	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT:
	NOTES: 1 2 3 4 5 6 7 8 9 10				



NODE: A32	TITLE: CONSTRUCT SYSTEM USING DRACO	NUMBER: DRA06

USED AT:	AUTHOR: James M. Neighbors PROJECT: Draco 1.0	DATE: 15-OCT-80 REV: 3	WORKING DRAFT RECOMMENDED PUBLICATION	READER DATE	CONTEXT:
	NOTES: 1 2 3 4 5 6 7 8 9 10				



1. all mechanisms are commands to the TFMREF subsystem

NODE: A323	TITLE: TRANSFORM AND REFINE INTERNAL FORM	NUMBER: DRA07